The Impact of Memory Models on Software Reliability in Multiprocessors

Alexander Jaffe
University of Washington
ajaffe@cs.washington.edu

Thomas Moscibroda
Microsoft Research
moscitho@microsoft.com

Laura Effinger-Dean
University of Washington
effinger@cs.washington.edu

Luis Ceze
University of Washington
luisceze@cs.washington.edu

Karin Strauss
Microsoft Research
kstrauss@microsoft.com

ABSTRACT

The memory consistency model is a fundamental system property characterizing a multiprocessor. The relative merits of strict versus relaxed memory models have been widely debated in terms of their impact on performance, hardware complexity and programmability. This paper adds a new dimension to this discussion: the impact of memory models on software reliability. By allowing some instructions to reorder, weak memory models may expand the window between critical memory operations. This can increase the chance of an undesirable thread-interleaving, thus allowing an otherwise-unlikely concurrency bug to manifest. To explore this phenomenon, we define and study a probabilistic model of shared-memory parallel programs that takes into account such reordering. We use this model to formally derive bounds on the *vulnerability* to concurrency bugs of different memory models. Our results show that for 2 concurrent threads, weaker memory models do indeed have a higher likelihood of allowing bugs. On the other hand, we show that as the number of parallel, buggy threads increases, the gap between the different memory models becomes proportionally insignificant, and thus the importance of using a strict memory model diminishes.

Categories and Subject Descriptors

F.1.2 [**Computation by Abstract Devices**]: Modes of Computation—*parallelism and concurrency*; G.3 [**Probability and Statistics**]: *Stochastic processes*; B.3.4 [**Memory Structures**]: Reliability, Testing, and Fault-Tolerance

General Terms

Theory, Reliability

Keywords

Memory consistency models, probabilistic analysis, sequential consistency, total store order, weak ordering, software reliability

PODC'11, June 6–8, 2011, San Jose, California, USA.

1. INTRODUCTION

A critically important property of a shared-memory multiprocessor is its *memory consistency model*. There has been an enormous amount of work on this subject, both in industry and academia. The memory consistency model describes which values may be returned by a load operation in a parallel or multi-threaded program. The strongest and most intuitive model is *Sequential Consistency* (SC) [15]. SC imposes two requirements on the execution of parallel programs: first, all processors must see the same *global order* of memory operations, and second, the operations for a particular processor must appear to execute in *program order*. This model is attractive for its high level of programmability, but the strict constraints on memory operation reordering rule out important optimizations such as access buffering, pipelining, or dynamic scheduling, which improve performance by hiding the latency of memory accesses. In order to enable these aggressive optimizations, a wide variety of *relaxed memory models* have been proposed. Relaxed memory models allow the reordering of certain types of memory operations at the cost of increased programming complexity, since programmers need to explicitly encode reordering restrictions to ensure correctness.

Historically, the vast literature on memory consistency models has discussed a three-way trade-off between performance, hardware complexity, and programmability. In this paper, we bring a new axis to this discussion: *software reliability*. Software is inherently unreliable, and is arguably becoming less reliable with pervasive concurrency. Concurrency bugs such as data races and deadlocks are extremely common in practice, and can cause unexpected failures in even production-level code.

In this paper, we investigate to what extent relaxed memory consistency models further contribute to the unreliability of parallel software by increasing the likelihood that concurrency bugs will manifest during an execution. For this purpose, we study a new probabilistic model for the instruction reordering introduced by relaxed memory models, and analyze a canonical buggy program (specifically, an atomicity violation [9, 4, 17]) with respect to this model. We compare three important memory consistency models: Sequential Consistency, Weak Ordering, and Total Store Order. We derive two interesting results for our model:

- We show that for 2 (or any small constant number of) parallel threads, the bug is indeed more likely to manifest under weaker memory models. This is intuitive and

follows from the following high-level argument: A typical concurrency bug, such as a data race, can manifest only during a short window of time. The reordering of operations caused by relaxed memory models may increase the size of this critical window, thus making the bug more likely to manifest. In the paper, we give precise bounds on this *vulnerability* of the three memory models.

- On the other hand, we show that as the number of parallel, buggy threads increases, the gap between the different memory models shrinks in proportion to the risk for even the strongest memory model. This implies that *as the number of parallel threads in the system increases, the importance of using a strict memory model diminishes* (with regard to the software reliability metric we study in this paper).

Notice that the latter result could have far-reaching implications on the choice of memory consistency models in future multi-core and massively parallel systems. Intuitively, one might expect that with more and more concurrent threads, stronger memory consistency models should be used in order to counter the generally increased likelihood of bugs. However, our results indicate that the opposite is the case: As the number of threads increase, the relative importance of having stronger memory models reduces to a minimum. The underlying reason is that the larger number of threads causes the likelihood that bugs occur to increase much more quickly than what even the strictest memory model is able to contain. That is, the asymptotic growth fundamentally works against using strict memory models as we increase the number of threads.

The technical content of our paper proceeds as follows. In Section 3, we introduce two distinct random processes, each of which is a natural object of inquiry in isolation. By combining them—treating the output of the first process as the input to the second—we model the end-to-end behavior of program execution. This allows us to answer our central question: how does the probability that a canonical data race manifests vary across memory models and quantity of threads?

The first process models the generation of a random program, and the subsequent randomized reordering of instructions. Specifically, in Section 4, we derive the probability that a certain essential window of vulnerability between two instructions widens. The second process enacts a random series of shifts on a set of heterogenous segments of the integer line. We use the positions of these line segments to model the interleaving of the vulnerable windows of the threads. In Section 5, we estimate the probability that each of these segments is shifted to mutually disjoint positions. Finally, the two processes are combined together in Section 6 to derive overall bounds on the probability of bug manifestation, first for two threads, then for a large number of threads. Due to lack of space, several proofs are omitted and deferred to the full version of this paper.

2. BACKGROUND & RELATED WORK

2.1 Memory Consistency Models

Memory models are a key aspect of the hardware/software interface in shared-memory multicore/multiprocessor systems. They determine what values read memory operations are allowed to return by dictating how memory operations are allowed to be reordered, as well as when writes become visible to other processors. They have major implications on the performance, design complexity and programmability of multiprocessor systems and the programs that run on them. Common misunderstandings about memory models often lead to bugs that are very difficult to find and fix, and can also lead to major performance issues. There exists a vast and rich line of literature on memory models (a good tutorial overview is presented in [1]). Most of the past work has focused on new memory models [11, 2, 13], hardware implementations [10, 12, 7], memory models for popular languages such as Java [18] and C++ [6], and compiler optimizations [16] and their relative merits [1, 5].

Relaxed memory models: The strongest memory model is Lamport's *Sequential Consistency* (SC) [15]. In order to enable important performance optimizations, a number of relaxed memory models have been proposed in the literature, with varying degrees of guarantees. One of the strongest examples is known as *Total Store Order* (TSO) [19]. In TSO, loads may execute before stores that precede them in program order, as long as no data dependency is violated. All other pairs of instructions must maintain strict program order. This model encapsulates the natural case in which stores are observed by remote processors in program order. Some stores may take extra time to be observed after their execution, but the local program is allowed to proceed. A similar, but slightly weaker consistency model is *Partial Store Order* (PSO) [19], which also allows the reordering of stores with respect to each other as long as they access distinct memory locations. A significantly weaker consistency model is *Weak Ordering* (WO) [8, 2]. The opposite extreme from Sequential Consistency, WO allows any memory operations to reorder with one another, as long as no data dependencies are violated. This model allows for an equal amount of optimization as a uniprocessor, but is also the most vulnerable to programmer error, since it requires explicit *fences* to prevent unwanted reorderings. Modern processors typically support relaxed models. For example, the x86 memory model [3, 14] supports a model similar to TSO and the IBM POWER architecture supports a form of WO.

The above memory consistency models follow a pattern: they can be defined by a subset of the four ordered memory operation pairs, specifying which pairs are allowed to reorder: For example, in the WO model, any two memory operations are allowed to be reordered; in SC, no two memory operations are allowed to be reordered; and in the TSO model, no two memory operations are allowed to be reordered, except that loads can reorder before stores (see Table 1).

Note that since in this paper we analyze a concurrency bug involving multiple threads, we ignore store atomicity [5], which is tangential to our present analysis. Moreover, we do not currently handle *fence* operations explicitly,[1] which are used to restrict reorderings and are typically used for synchronization. For that reason, we do not consider models such as Release Consistency (RC) [11], which differs mainly in the types of fences supported. As we discuss in Section 7, it will be interesting to extend our process to distinguish such memory models.

[1]However, our shift process in Section 5 can be used to simulate a behavior similar to that arising from the use of fences.

ST/ST	ST/LD	LD/ST	LD/LD	Name
				Sequential Consistency
	X			Total Store Order
X	X			Partial Store Order
X	X	X	X	Weak Ordering

Table 1: Important memory models. A "X" in column ST/LD means that the ordering restriction from stores to later loads can be relaxed, i.e., loads can complete before stores that precede them in program order. With regard to our model in Section 3.1.2, this means that a LD can settle past (swap with) a preceding ST. Other columns are analogous.

2.2 Race Conditions

A common type of bug in shared-memory multithreaded programming is a *race condition*, which occurs when correctness depends on an assumption about the order in which instructions from two or more threads interleave. In particular, an *atomicity violation* [9] occurs when the programmer assumes that multiple instructions will execute as an atomic unit, but fails to insert the proper synchronization. A recent study showed that atomicity violations are extremely common in "real world" programs [17]. Race conditions are often difficult to identify due to nondeterminism: the program may behave correctly most runs, but fails only for specific thread interleavings.

A canonical example of an atomicity violation is as follows:

Thread 1	Thread 2
1: int loc = x;	1: int loc = x;
2: loc = loc + 1;	2: loc = loc + 1;
3: x = loc;	3: x = loc;

Here x is a shared variable (with x = 0 initially) and loc is local to each thread. Two threads simultaneously try to increment x by loading its value into a local variable, incrementing that local variable, then storing the updated value back to x. The programmer's intent is that x = 2 after both threads finish executing. However, the program has a race condition that can result in the spurious outcome x = 1. For instance, suppose that the two threads interleave as follows: (1) Thread 1 executes Lines 1 and 2; (2) Thread 2 executes Lines 1 and 2; (3) Thread 1 executes Line 3; (4) Thread 2 executes Line 3. This interleaving produces the final result x = 1. We say that the bug *manifests* because the result did not match programmer intent.

The standard solution for race conditions like the example above is to protect the variable x with a lock. However, locking protocols can be extremely complicated in large programs, and in practice, a concurrency bug may easily slip past even the most experienced programmers. Note that such bugs can manifest in any memory model, even Sequential Consistency.

3. MODEL

Our goal is to study how the use of different memory models impacts the likelihood of an error occurring given a canonical atomicity violation. In this section, we describe a model that allows us to formally analyze these likelihoods. It is a probabilistic model of parallel program executions under memory models that may permit reordering. At a high level, we consider two or more threads which execute a simple program containing an atomicity bug. The program consists of basic memory operations (stores and loads). Depending on the memory model under consideration, the operations in each thread are then independently reordered via a random process we call the *settling process*. Finally, we use a thread interleaving model—the *shift model*— to model the execution of the program by interleaving the instructions of different threads. The probability of the bug manifesting is determined by analyzing how the operations from the threads interleave. We show in this paper that, when executing two threads, this probability crucially depends on the underlying memory model. Yet, perhaps counter-intuitively, we show that as the number of threads grows larger, the relative difference between the memory models becomes smaller and smaller.

3.1 Program Model

We first describe a process for modeling a typical, randomly reordered program. The process proceeds in two phases: program generation and program reordering.

3.1.1 Program Generation

We model an initial program based on the canonical atomicity violation bug described in §2.2. The program is a sequence S of memory operations $x_1, x_2, \ldots, x_m, x_{m+1}, x_{m+2}$, where each x_i has *type* $\tau(x_i) \in \{\mathsf{LD}, \mathsf{ST}\}$. x_{m+1} and x_{m+2} are Lines 1 and 3 of the canonical bug, respectively. Since we are only concerned with memory operations, we omit Line 2 (which accesses only the local variable loc), and we will use the terms *instruction* and *memory operation* synonymously in this paper. We assume for simplicity that that only x_{m+1} and x_{m+2} access the same location.[2] We will call x_{m+1} the *critical load* and x_{m+2} the *critical store*. An *initial program order* S_0 starts with a random sequence of m independently distributed LD and ST operations; $\tau(x_i) = \mathsf{ST}$ with probability p and LD with probability $1 - p$. Furthermore, for convenience in the analysis, it will be useful to approximate a very long program by letting $m \to \infty$.

3.1.2 Instruction Reordering: The Settling Process

Different memory models allow for different forms of instruction reorderings. We model this relaxation of program order using a probabilistic *settling process*. This random process models instruction reordering by taking a (random) initial program order as input, and producing a reordering of that initial program. The settling process takes into account which kinds of reorderings are allowed by the memory consistency model under consideration, and generates a random program order that is allowed to occur given the kinds of reorderings. In this section, we give an informal description of the settling process; a formal definition is given in the full paper. Figure 1 presents a visualization of the settling process.

Given an initial program order S_0, the settling process proceeds in $m+2$ rounds. In the rth round, (1) the program order S_{r-1} from the end of the $(r-1)$st round is taken as the input, and (2) the rth instruction is *settled* in this program order, which (3) creates the new program order S_r. The final output of the settling process is the program order S_{m+2} after settling the critical store x_{m+2}. Settling the rth instruction in round r of the process works as follows. Instruction x_r is recursively reordered (that is, swapped in

[2]If two instructions access the same location, they cannot reorder, so this assumption simplifies our analysis.

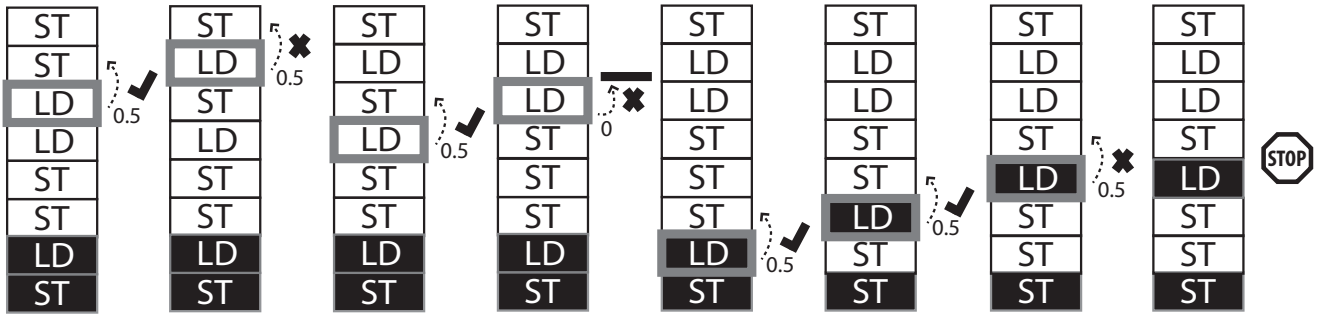

Figure 1: An instantiation of the settling process under TSO. LDs repeatedly settle upward with probability 1/2. If they fail to settle, or encounter another LD, they stop permanently, and the next-lowest LD begins. The black boxes represent the critical instructions. The grey outlines indicate the currently settling instruction. The bottom four instructions in the final order form the *critical window*.

the current program order) with its preceding instruction (initially, this is the instruction at position $r-1$), until a reordering "fails," in which case x_r remains at its current position in the program order. A reordering always fails if the memory consistency model does not allow two operations of this type to be reordered. Otherwise, the reordering succeeds with some fixed probability s, and fails with probability $1-s$.[3] When a reordering fails, we move onto the next round.

For ease of exposition, we will set both probabilities p (from program model) and s to be $1/2$ in subsequent sections. However, note that as long as s and p are constant, the key theorems and conclusions derived in this paper remain fundamentally the same (though some of the numerical values change somewhat).

Examples: In SC, no instructions are allowed to be reordered; hence $S_{m+2} = S_0$. In WO, all types of reorderings are allowed, so, starting from instruction 2 in the initial program order, each instruction is settled using a series of swaps with its preceding instructions, until with probability $1-s$ a swap fails. Then the next instruction is settled, and so forth. TSO relaxes only the ST $\rightarrow$ LD ordering, which in our model implies that a LD may reorder with a preceding ST with probability s, but all other types of reorderings fail.

We will represent the result of a settling process by a permutation on the indices. For thread k, $\pi^{(k)}(i) : [1, 2, \ldots, m+2] \rightarrow [1, 2, \ldots, m+2]$ maps the instruction starting at position i to its final settled position.

The settling process has two key features: (1) memory model constraints are enforced (two operations can reorder only if allowed by the memory model), and (2) reorderings that *are* allowed occur with a fixed likelihood. One effect of the latter property is that in the final program order, most instructions will not to move too far from their position in the initial program order. The critical property of a memory consistency model that we seek to capture is the *degree to which individual instructions can reorder beyond other instructions, and thus move further away from their original position.*

[3]A more general form of the settling model allows different nonzero probabilities for different kinds of reorderings, depending on the types of memory operations involved. For example $s_{\mathsf{LD},\mathsf{LD}}$ can be different from $s_{\mathsf{LD},\mathsf{ST}}$, even if both are nonzero.

3.2 Thread Interleaving Model

We describe a second high-level random process, which is used to determine the interleaving of n threads when they are executed simultaneously on a multiprocessor. In fact, the process is quite general, and may be of independent interest as a probabilistic model. We first describe it in the abstract, then discuss how it will be used to determine the effect of the program model's output on the probability of bug manifestation.

Definition 1. Consider a sequence of n positive line segments originating at 0, having integer lengths $\bar{\gamma} = \gamma_1, \ldots, \gamma_n$. A *shift process* translates the segments by i.i.d. geometric random variables $s_1, \ldots, s_n$. Then the random event of interest, called $A(\bar{\gamma})$, is the event that the segments are shifted such that all are mutually disjoint. That is,

$$A(\bar{\gamma}) := [s_i, s_i + \gamma_i] \cap [s_j, s_j + \gamma_j] = \emptyset \ \forall \ i \neq j.$$

In Section 5, we will analyze the probability of $A(\bar{\gamma})$ for arbitrary segment lengths $\bar{\gamma}$. However, to connect this model to the task at hand, we will go on to think of these segment lengths as the *critical windows* of reordered programs generated by the program model.

Recall that we study a canonical data race, for which correct execution requires that each thread's pair of critical LD and critical ST be executed atomically. We thus refer to the sequence of instructions between the critical LD and ST (inclusively) as the *critical window* of a thread. We let B_γ^k be the event that the final ordering of thread T_k inserts γ instructions between the critical LD and ST, (sometimes referred to as the *critical window growth* of a memory model). Manifestation of the data race corresponds exactly to the event that when the reordered threads are executed in parallel, some pair of critical windows are *not* executed disjointly. We let A refer to the event that critical windows are disjoint. One can then think of $\Pr[B_\gamma^k]$ and $\Pr[A]$ as the two fundamental values we seek to characterize in this paper - each a measure of the vulnerability of a memory model to this canonical data race.

The shift model is used to simulate the parallel execution of the critical windows of each thread, under the following assumptions. All threads are assumed to initially be identical copies of a single program, generated randomly as in Section 3.1.1. Each thread is then independently reordered according the process of Section 3.1.2. We then simulate

the parallel execution of the reordered threads by placing the final instruction of each critical window the origin of the number line (here representing time in reverse, with 0 being the final time step of execution), and using the shift model of Definition 1 to model the *varying rates of execution* of each thread. After shifting, the execution of each instruction is assumed to take one unit of time; instructions begin and end synchronously across all threads, in lock-step. We assume that instructions instantaneously read the current state of the system at the beginning of the time step, and instantaneously commit their changes at the end of the time step. In this way we ensure a clear semantics for the state of the system at any given time: when a LD executes, it observes all the effects of any ST that completed in a time step preceding it.

We can now observe the circumstances in which a data race manifests. There must be two threads such that, subsequent to reordering, the final regions of time steps between the critical LD and ST (inclusive) overlap with one another. In this case the data race must manifest, because one of the LDs must observe a value after (or simultaneous to) the other LD being observed, but before the other ST has committed.

A formal definition and a graphical visualization of the shift process is in the full paper.

4. THE CRITICAL WINDOW

In this section, we study what is perhaps the core component of our random process, and the only one that directly distinguishes the memory models: the reordering of instructions within an individual thread. In particular, we are interested in the final distribution of the size of the *critical window* between the critical LD and ST. For the extreme memory models of Sequential Consistency and Weak Ordering, we are easily able to exactly characterize this distribution. The bulk of the technical challenge of this section (and consequently of later sections) is in establishing results for the more subtle model, Total Store Order. By carefully conditioning on several auxiliary random variables, lower bounding complex algebraic terms by their low-indexed values, and utilizing a bound on the *partition number* of certain integers, we derive rather sharp approximations for the distribution of the critical window size. These bounds will in subsequent sections be plugged into derived formulae for the probability of bug manifestation, as a function of the thread *interleaving* process. Though the results in this section are tailored specifically to the thread generation and reordering processes specified in the previous section, it is worthwhile to observe how the asymptotics of the overall bug manifestation probability will not depend delicately on the details of this process.

We will be estimating the critical window growth, $\Pr[B_\gamma^k]$, for a select set of memory models. Recall that B_γ^k is the event that the thread T_k inserts γ instructions between the critical LD and ST in reordering. Because we will be considering a single fixed thread in this subsection, we will refer to the event B_γ^k by B_γ, and the permutation $\pi^{(k)}$ by π. The first two memory models can be considered a warmup, for the substantially more challenging case of Total Store Order. All of these results are captured in the following theorem.

THEOREM 4.1. *The critical window growth behaves according to the following functions:*

- ***Sequential Consistency:***

$$\Pr[B_\gamma] = \begin{cases} 1 & \text{if } \gamma = 0, \\ 0 & \text{if } \gamma > 0. \end{cases}$$

- ***Weak Ordering:***

$$\Pr[B_\gamma] = \begin{cases} 2/3 & \text{if } \gamma = 0, \\ (2^{-\gamma})/3 & \text{if } \gamma > 0. \end{cases}$$

- ***Total Store Order:***

$$\Pr[B_\gamma] = \begin{cases} 2/3 & \text{if } \gamma = 0, \\ (6/7) \cdot 4^{-\gamma} + R(\gamma) \cdot 2^{-\gamma} & \text{if } \gamma > 0, \end{cases}$$

for non-negative approximation term $R(\gamma) \leq \frac{2}{21}$.

Observe that the critical window grows at vastly different rates across the models. Up to lower-order terms, the probability of a window size γ is $2^{-\gamma}$ in Weak Ordering, $(2^{-\gamma})^2$ in Total Store Order, and 0 in Sequential Consistency. It remains to be seen in later sections the extent to which this window size effects bug manifestation.

PROOF (THEOREM 4.1—SEQUENTIAL CONSISTENCY). Under sequential consistency, no instruction is ever allowed to reorder. Hence $\Pr[B_0] = 1$, and $\Pr[B_\gamma] = 0 \ \forall \gamma \neq 0$. □

We next consider the case of intermediate difficulty: Weak Ordering.

PROOF (THEOREM 4.1—WEAK ORDERING). Under weak ordering, all four ordered pairs of instruction types are allowed to pass one another. Recall that we assume a strong normal form, in which all possible swaps occur with probability 1/2. Hence in weak ordering, each subsequent instruction continually moves up with probability 1/2, until it ever fails to swap. This applies to the critical load and critical store as well, with the exception that the critical store will never pass the critical load, (because they access the same address). To calculate the probability, we condition on the resting position of the critical LD, which entails a given resting position for the critical ST, for any $\gamma > 0$.

$$\begin{aligned} \Pr[B_\gamma] &= \Pr[\pi(m+2) - \pi(m+1) = \gamma + 1] \\ &= \sum_{i=\gamma}^{\infty} \Pr[\pi(m+1) = m+1-i] \\ &\qquad \cdot \Pr[\pi(m+2) = m+2-i+\gamma \mid \\ &\qquad\qquad \pi(m+1) = m+1-i] \\ &= \sum_{i=\gamma}^{\infty} 2^{-(i+1)} 2^{-(i+1-\gamma)} \ = \ \frac{2^{-\gamma}}{3}. \end{aligned}$$

We must handle the case of $\gamma = 0$ separately, because here the critical ST stops moving "automatically," when it runs up against the critical LD.

$$\begin{aligned} \Pr[B_0] &= \sum_{i=0}^{\infty} \Pr[\pi(m+1) = m+1-i] \\ &\qquad \cdot \Pr[\pi(m+2) = m+2-i \mid \pi(m+1) = m+1-i] \\ &= \sum_{i=0}^{\infty} 2^{-(i+1)} 2^{-(i)} \ = \ 2/3. \quad \square \end{aligned}$$

Finally we turn to the far more challenging setting of Total Store Order.

Proof (Theorem 4.1—Total Store Order).
One of the strongest and most commonly used relaxed memory models, Total Store Order (TSO) only permits loads to swap with stores. Hence in calculating the distribution of window size, we need only consider the number of stores located directly before the critical load. Those stores will never move themselves, and the critical load can never swap past the first load above it. Moreover, the critical store never swaps with anything, so its final position is fixed.

However, deriving bounds on $\Pr[B_\gamma]$ is difficult. LD *operations may reorder past* ST *operations, thus pushing longer sequences of* ST *operations together.* In this section we derive bounds on the critical window growth for TSO, which is a core technical contribution of this paper. The proof is quite involved. Much difficulty arises in gaining control over the relative positions of LDs and STs. We outline the steps taken to estimate the critical window growth below. The majority of these steps are non-trivial, and often involve a delicate case analyses.

Proof Outline.

1. Express the critical window probability in terms of a series of new random variables, L_μ: the event that the second-to-last reordering leaves exactly μ contiguous STs above the critical LD.
2. To calculate the probability of L_μ, condition on the value of another series of random variables, Ψ_μ: the number of LDs *initially* between the critical LD and the $\mu+1$th lowest ST.
3. Express the Ψ_μ-conditioned probability of L_μ in terms of the limit of the fraction of STs near the bottom of a reordered thread, and another probability, $\Pr[F_\mu|\Psi_\mu = q]$: the chance of q LDs all reordering out of a region of *at least* μ STs.
4. To estimate $\Pr[F_\mu|\Psi_\mu = q]$, condition on a new random variable, Δ: the sum, over STs, of the number of LDs below each ST. Express the probability of Δ in terms of the weighted sum of several integer *partition numbers*, and estimate these by a simple lower bound.
5. After combining the above elements to bound the probability of L_μ, lower bound an ugly term of this expression by its value at $\mu = 1$, checking via the derivative that this term is increasing in μ.
6. Use the lower bound on the probability of L_μ to finally lower bound the probability of a given window size. To achieve an upper bound, calculate the total probability not attributed to some L_μ in the lower bound, and attribute it to the worst-possible case.

We now move on to execute this plan in detail.

Step 1—Number of contiguous STs above the critical LD: Recall that S_0 (S_{m+2}) denotes the initial (final) instruction order, and that S_m refers to the instruction order just *before* the critical load is settled. For convenience, we define the following basic random events. Let $S_{\mathsf{LD},i}(j)$ be the event that after the jth instruction of S_i is a LD. Furthermore, we define $S_{\mathsf{LD},i}(j,k) = \bigwedge_{\ell=j}^{k} S_{\mathsf{LD},i}(\ell)$ as the event that the entire contiguous range from j to k in S_i consists of LDs. $S_{\mathsf{ST},i}(j)$ and $S_{\mathsf{ST},i}(j,k)$ are defined accordingly.

For $\mu \in \mathbb{N}$, we define L_μ as the event that in S_m, there are exactly μ ST operations immediately preceding the critical LD. In other words,

$$L_\mu = S_{\mathsf{LD},m}(m-\mu) \wedge S_{\mathsf{ST},m}(m-\mu+1, m).$$

The critical LD may only move γ positions if there are at least γ contiguous ST operations above it. Hence for any γ, we have

$$\Pr[B_\gamma] = \sum_{\mu=\gamma}^{\infty} \Pr[B_\gamma|L_\mu] \cdot \Pr[L_\mu].$$

Deriving $\Pr[B_\gamma|L_\mu]$ is straightforward. If $\mu = \gamma$, we have $\Pr[B_\gamma|L_\gamma] = 2^{-\gamma}$, as the critical LD must pass all γ STs. After that, it stops because the next instruction is a LD. For $\mu > \gamma$, we have $\Pr[B_\gamma|L_\mu] = 2^{-(\gamma+1)}$, because the instruction above the γth ST is also a ST. Hence there is only a $1/2$ probability of the reordering completing when it reaches that point.

It remains to derive bounds for $\Pr[L_\mu]$ for all μ. This is the primary technical lemma of the proof.

Lemma 4.2. *For any $\mu > 0$, $\Pr[L_\mu] \geq \frac{4}{7} \cdot 2^{-\mu}$. Moreover, $\Pr[L_0] = 1/3$ exactly.*

Proof. We will approach this lemma by asking (1) how many LDs are interspersed among the first μ STs above the critical LD, and (2) what is the probability that all of those LDs settle such that we are left with μ contiguous STs above the critical LD. Because STs cannot settle past LDs in this model, nothing happens during rounds in which a ST can move; the technical difficulty arises in the motion of the LDs.

Step 2—Number of interspersed LDs: In the initial program order S_0, let Φ_μ refer to the position of the μth-lowest non-critical ST. Formally,

$$\Phi_\mu = \min\{i : |\{j \geq i : S_{\mathsf{ST},0}(j)\}| = \mu + 1\}.$$

Furthermore, let Ψ_μ refer to the number of LD operations above the critical LD but below the μth-lowest non-critical ST. That is,

$$\Psi_\mu = m + 1 - \mu - \Phi_\mu.$$

Note that as the program length goes to infinity, the probability that such a Φ_μ and Ψ_μ exist goes to 1. Now we can express $\Pr[L_\mu]$ as

$$\Pr[L_\mu] = \sum_{q=0}^{\infty} \Pr[L_\mu|\Psi_\mu = q] \cdot \Pr[\Psi_\mu = q]. \tag{1}$$

We have $\Pr[\Psi_\mu = q] = 2^{-\mu}2^{-q}\binom{\mu+q-1}{q}$ because there are $\binom{\mu+q-1}{q}$ ways to build a string of μ STs and q LDs such that the top instruction is a ST.

Step 3—Probability of interspersed LDs settling out: The difficult part of bound (1) is $\Pr[L_\mu|\Psi_\mu = q]$. This is the probability that

(A) All q LDs between the ST at Φ_μ and the critical LD settle up until they pass the ST at Φ_μ,

(B) but do not settle so far that the settled instruction above the ST at Φ_μ is another ST.

(B) is due to the fact that L_μ specifies that there be *exactly* μ STs above the critical LD. The probability of (B) relies

on the instruction directly above Φ_μ in $S_{\Phi_\mu-1}$. If it is a LD, then (B) holds automatically, since all the LDs must stop settling. However, if it is a ST, then (B) only holds if not all of the q LDs that have passed the ST at Φ_μ also pass the next-highest ST. Hence this is the first property on which we condition.

$$\Pr[L_\mu|\Psi_\mu = q] = \Pr[L_\mu \wedge S_{LD,\Phi_\mu-1}(\Phi_\mu - 1)|\Psi_\mu = q] + \Pr[L_\mu \wedge S_{ST,\Phi_\mu-1}(\Phi_\mu - 1)|\Psi_\mu = q].$$

By Bayes' Law,

$$\Pr[L_\mu \wedge S_{LD,\Phi_\mu-1}(\Phi_\mu - 1)|\Psi_\mu = q] = \Pr[S_{LD,\Phi_\mu-1}(\Phi_\mu - 1)|\Psi_\mu = q] \cdot \Pr[L_\mu|S_{LD,\Phi_\mu-1}(\Phi_\mu - 1) \wedge \Psi_\mu = q].$$

We first consider the latter term. Because the final instruction that settles above Φ_μ will be a LD under these conditions, this depends only on the bottom μ instructions settled above the critical LD being STs. For shorthand, let

$$F_\mu = S_{ST,m}(m - \mu + 1, m).$$

Then

$$\Pr[L_\mu|S_{LD,\Phi_\mu-1}(\Phi_\mu - 1) \wedge \Psi_\mu = q] = \Pr[F_\mu|\Psi_\mu = q].$$

In contrast, for L_μ to hold given $S_{ST,\Phi_\mu-1}(\Phi_\mu - 1)$, it does not suffice for the q LDs to move past Φ_μ. They must also not all settle past the next highest instruction. They do so with probability 2^{-q}. Hence

$$\Pr[L_\mu|S_{ST,\Phi_\mu-1}(\Phi_\mu - 1) \wedge \Psi_\mu = q] = \Pr[F_\mu|\Psi_\mu = q] \cdot (1 - 2^{-q}).$$

Putting these expressions together, we find that

$$\begin{aligned}&\Pr[L_\mu|\Psi_\mu = q]\\ &= \Pr[F_\mu|\Psi_\mu = q] \cdot \Pr[S_{LD,\Phi_\mu-1}(\Phi_\mu - 1)]\\ &\quad + \Pr[F_\mu|\Psi_\mu = q] \cdot \Pr[S_{ST,\Phi_\mu-1}(\Phi_\mu - 1)] \cdot (1 - 2^{-q})\\ &= \Pr[F_\mu|\Psi_\mu = q] \cdot (1 - 2^{-q} \cdot (1 - \Pr[S_{ST,\Phi_\mu-1}(\Phi_\mu - 1)])).\end{aligned}$$

We first derive an exact value for $\Pr[S_{ST,i}(i)]$. Though it is difficult to determine the probability that a given instruction is a ST in general, this particular value can be derived exactly through a recurrence relation.

Claim 4.3.

$$\lim_{i\to\infty} \Pr[S_{ST,i}(i)] = 2/3.$$

Proof. After reordering stage i, instruction i can be a ST in one of two ways. Either it can initially be a ST, (in which case it never reorders) or it can initially be a LD, the instruction above it can be settled as a ST, and the two can swap. Hence

$$\Pr[S_{ST,i}(i)] = \frac{1}{2} + \frac{1}{2} \cdot \Pr[S_{ST,i-1}(i-1)] \cdot \frac{1}{2}.$$

This is a recurrence relation of the form $X_i = b + aX_i$, which has the solution $X_i = \frac{b}{1-a} + a^{i-1}(X_1 - \frac{b}{1-a})$. Plugging in $X_1 = 1/2$, $a = 1/4$, $b = 1/2$, we find

$$\begin{aligned}\Pr[S_{ST,i}(i)] &= \frac{1/2}{1-1/4} + (1/4)^{i-1}\left(1/2 - \frac{1/2}{1-1/4}\right).\\ &= 2/3 + (1/4)^{i-1}(1/2 - 2/3)\end{aligned}$$

The resulting probability is a function of i, but we are interested in the steady-state as the size of the program goes to infinity. Hence the second term falls out.

$$\lim_{i\to\infty} \Pr[S_{ST,i}(i)] = 2/3. \quad \square$$

Now that we know the typical fraction of instructions near the bottom of the program that are STs after reordering, we can derive a bound on $\Pr[F_\mu|\Psi_\mu = q]$.

Step 4—Estimating $\Pr[F_\mu|\Psi_\mu = q]$:

Claim 4.4.

$$\Pr[F_\mu|\Psi_\mu = q] \geq \frac{2^{-(q-1)} - 2^{-\mu q}}{\binom{\mu+q-1}{q}}.$$

Proof. Everything in this proof is implicitly conditioned on the event $\Psi_\mu = q$. Let the random variable

$$\Delta = \sum_{\Phi_\mu < i \leq m:\tau_{\mathsf{LD},0}(i)} |\{\Phi_\mu \leq j < i : \tau_{\mathsf{ST},0}(j)\}|$$

represent the total number of positions that LDs from Φ_μ to m must move up, in order to leave a sequence of μ STs immediately above the critical LD. It must be that $\Delta \geq q$, because at least instruction Φ_μ is a ST, and $\Delta \leq \mu q$, because no LD can be required to pass more than μ STs. With this definition, we may write $\Pr[F_\mu|\Psi_\mu = q] = \sum_{\delta=q}^{\mu q} \Pr[\Delta = \delta] \cdot 2^{-\delta}$. The exact value of $\Pr[\Delta = \delta]$ can be stated formally, but not in a closed form. Namely, let $\phi(x, y, z)$ be the number of distinct multi-sets of y positive integers summing to x, such that each integer is at most z. This is a variant on the much-studied *partition number* of x. Then $\phi(\delta, q, \mu)$ is exactly the number of arrangements of q LDs and μ STs (beginning with a ST) such that δ is the sum of the number of STs above each of the LDs. (For each LD, we simply select how many STs to place it below—the relative order of the LDs is immaterial.) There are $\binom{m+q-1}{q}$ total arrangements of LDs and STs beginning with a ST. Hence

$$\Pr[\Delta = \delta] = \frac{\phi(\delta, q, \mu)}{\binom{\mu+q-1}{q}},$$

and

$$\Pr[F_\mu|\Psi_\mu = q] = \sum_{\delta=q}^{\mu q} \frac{\phi(\delta, q, \mu)}{\binom{\mu+q-1}{q}} \cdot 2^{-\delta}.$$

Simple forms for $\phi(x, y, z)$ are not known. Asymptotic results exist, but are not helpful here because the terms with small parameters have the largest contributions. However, to achieve a good bound it suffices to show that $\phi(\delta, q, \mu) \geq 1$ when $q \leq \delta \leq \mu q$. To show that a partition exists that achieves any number in this range, consider the following construction. Set $\delta \bmod q$ of the integers to $\lceil \delta/q \rceil$, and set the rest of the integers to $\lfloor \delta/q \rfloor$. We can set the integers this large, because $\delta/q \leq (\mu q)/q = \mu$. Then the chosen integers sum to $(\delta \bmod q)\lceil \delta/q \rceil + (q - (\delta \bmod q))\lfloor \delta/q \rfloor$ which can be shown to be exactly δ. Hence we may write

$$\Pr[F_\mu|\Psi_\mu = q] \geq \frac{1}{\binom{\mu+q-1}{q}} \sum_{\delta=q}^{\mu q} 2^{-\delta} = \frac{2^{-(q-1)} - 2^{-\mu q}}{\binom{\mu+q-1}{q}}. \quad \square$$

Having derived a bound for $\Pr[F_\mu|\Psi_\mu = q]$, we are now in a position to conclude the proof of Lemma 4.2. First note

that $\Pr[L_0] = 1/3$, by Claim 4.3. For values of μ greater than 0, Claim 4.4 will be the central tool in the proof, which is left to the full version of the paper. □

The remainder of the proof of Theorem 4.1, steps 5 and 6, is deferred to the full version of the paper. □

5. SHIFT PROCESS

Here we discuss the next component of our analysis: a "shift process" meant to capture the interleaving of reordered threads. We refer the reader back to the definition in Section 3.2. This process is where the critical windows derived from the reordering process come into effect.

In the analysis that follows, we assume that each critical window's shift is distributed geometrically, representing the intuition that threads are exponentially less likely to execute at progressively increasing offsets from one another. Let $\bar{\gamma} = (\gamma_1, \gamma_2, \ldots, \gamma_n) \in \mathbb{N}^n$ be a sequence of integral "segment lengths." In subsequent sections, γ_k will be used to represent the length of the critical window of thread T_k. We define a shift process on $\bar{\gamma}$ as follows. Consider n segments of the line, of lengths $\gamma_1, \gamma_2, \ldots, \gamma_n$, and let the starting point of each segment be shifted up from 0 by an i.i.d. positive random variable s_i. We are interested in the probability that the resulting set of shifted segments is non-overlapping. In other words, we would like to bound $\Pr[A(\bar{\gamma})]$, where $A(\bar{\gamma})$ is the event that $\forall i \neq j \in \{1, 2, \ldots, n\}$, we have $[s_i, s_i + \gamma_i] \cap [s_j, s_j + \gamma_j] = \emptyset$.

The following theorem states this probability precisely, and as such is not particularly enlightening on its own. However, when the segment lengths are random variables drawn from a well-understood distribution (as they are in the case of reordered random threads), we will be able to state the probability concisely.

THEOREM 5.1.

$$\Pr[A(\bar{\gamma})] = \frac{2^{-(\binom{n+1}{2}-1)}}{\prod_{i=1}^{n-1}(1-2^{-(n+1-i)})} \sum_{\sigma \in Sym_n} \prod_{i=1}^{n-1} 2^{-(n-i)\gamma_{\sigma(i)}},$$

where Sym_n is the symmetric group of degree n: the set of all permutations on n elements.

The following corollary simplifies this expression:

COROLLARY 5.2. *For some $c(n) \in [2, 4]$,*

$$\Pr[A(\bar{\gamma})] = c(n) \cdot 2^{-\binom{n+1}{2}} \cdot \sum_{\sigma \in Sym_n} \prod_{i=1}^{n-1} 2^{-(n-i)\gamma_{\sigma(i)}}.$$

In particular, $c(2) = \frac{8}{3}$ exactly.

The proof of the corollary is in the full version of the paper. We now turn to the proof of the main theorem. The challenge is to characterize the probability that the next segment is shifted to a position disjoint from all previous segments. At first glance, it is difficult to handle the huge and diverse set of legal placements for a set of segments. Our key insight is to condition on the *relative order* of the magnitude of the shifts. We then consider the probability that each segment is disjoint from the previous threads in this order. In so doing, we are able to exploit the memorylessness of the geometric distribution. Let t be an arbitrary segment, and t' be the segment immediately preceding it in this order. To understand the distribution of disjoint placements for t, we need only know the distribution of the origin of t'. Then by assuming that the segments are disjoint, we can infer that the origin of t is distributed according the origin of t', plus the length of t', plus an independent geometric random variable.

PROOF (THEOREM 5.1). Let s_i be a geometric random variable with expectation 2 (i.e., $s_i = k$ with probability $2^{-(k+1)}$ $\forall k \in \mathbb{N}$). In order to analyze the probability of $A(\bar{\gamma})$, we will take the following steps. We will first condition on the ordering of the segments. Then for a given ordering, we will use the memorylessness of the shift variables to calculate the probability of each successive segment being disjoint from each previous.

For a permutation σ on $\{1, 2, \ldots, n\}$, let Y_σ be the event that for all i, the ith largest shift occurs on segment $\sigma(i)$. That is, $s_{\sigma(1)} \geq s_{\sigma(2)} \geq \cdots \geq s_{\sigma(n)}$. Then $\Pr[A(\bar{\gamma})] = \sum_{\sigma \in Sym_n} \Pr[A(\bar{\gamma}) \wedge Y_\sigma]$.

We now analyze $\Pr[A(\bar{\gamma}) \wedge Y_\sigma]$. We will refer to this event by $A(\bar{\gamma}, \sigma)$. For all segments to be disjoint, it must be the case that each segment begins after the *end* of every segment that began before it. σ captures exactly the order in which segments begin. So disjointness means that for all i, j s.t. $\sigma(j) > \sigma(i)$, segment j begins after the end of segment i. Hence for each i, we may condition on the shift of the segment with the ith largest shift, and consider the probability that each segment with a smaller shift follows its completion.

$$\begin{aligned}
\Pr[A(\bar{\gamma}, \sigma)] &= \sum_{\ell_1=0}^{\infty} \Pr[A(\bar{\gamma}, \sigma) \wedge s_{\sigma(1)} = \ell_1] \\
&= \sum_{\ell_1=0}^{\infty} \Pr[A(\bar{\gamma}, \sigma) \wedge s_{\sigma(1)} = \ell_1 \wedge \bigwedge_{i=2}^{n} s_{\sigma(i)} \geq \ell_1 + \gamma_{\sigma(1)}] \\
&= \sum_{\ell_1=0}^{\infty} \Pr[A(\bar{\gamma}, \sigma) | s_{\sigma(1)} = \ell_1 \wedge \bigwedge_{i=2}^{n} s_{\sigma(i)} \geq \ell_1 + \gamma_{\sigma(1)}] \\
&\qquad \cdot \Pr[s_{\sigma(1)} = \ell_1] \cdot \prod_{i=2}^{n} \Pr[s_{\sigma(i)} \geq \ell_1 + \gamma_{\sigma(1)}].
\end{aligned}$$

The third equality is due to the independence of the shift variables. Let $\bar{\gamma}^i$ refer to the restriction of $\bar{\gamma}$ to the segment indices with the $n - i + 1$ smallest shifts (i.e., $\bar{\gamma}^i = \bar{\gamma}_{|[n] \setminus \bigcup_{j=i}^{n} \sigma(j)}$). Similarly, let σ^i refer to the restriction of σ to the $n - i + 1$ smallest shifts (i.e., $\sigma^i = \sigma_{|[n] \setminus [i-1]}$). We define these structures so that we can express the disjointness event in terms of a new disjointness event on a smaller set of unconditioned segments. In particular, let $A(\bar{\gamma}^i, \sigma^i)$ be the disjointness event for an independent random shift process on segments $\sigma(i), \sigma(i+1), \ldots, \sigma(n)$, with permutation σ^i pointing to the new indices of these segments. We will see that we are permitted to condition on such a prior event, because of the memoryless of the shift variables.

Conditioned on the first segment being disjoint from all the following segments, we need only consider the event $A(\bar{\gamma}^2, \sigma)$. Then due to the memorylessness of the shifts, we have

$$\begin{aligned}
&\Pr[A(\bar{\gamma}^i, \sigma^i) | s_{\sigma^i(1)} = \ell_1 \wedge \bigwedge_{j=2}^{n} s_{\sigma^i(j)} \geq \ell_1 + \gamma_{\sigma^i(1)}] \\
&\qquad = \Pr[A(\bar{\gamma}^{i+1}, \sigma^{i+1}) | \bigwedge_{j=2}^{n} s_{\sigma^i(j)} \geq \ell_1 + \gamma_{\sigma^i(1)}]
\end{aligned}$$

$$= \Pr[A(\bar{\gamma}^{i+1}, \sigma^{i+1}) | \bigwedge_{j=2}^{n} s_{\sigma^i(j)} \geq 0] = \Pr[A(\bar{\gamma}^{i+1}, \sigma^{i+1})].$$

We now observe a simple recurrence relation that defines $\Pr[A(\bar{\gamma}^i, \sigma^i)]$.

$$\Pr[A(\bar{\gamma}^i, \sigma^i)] = \sum_{\ell_1=0}^{\infty} \Pr[A(\bar{\gamma}^{i+1}, \sigma^{i+1})] \cdot \Pr[s_{\sigma^i(1)} = \ell_1] \cdot \prod_{j=i+1}^{n} \Pr[s_{\sigma^i(j)} \geq \ell_1 + \gamma_{\sigma^i(1)}]$$

$$= \sum_{\ell_1=0}^{\infty} \Pr[A(\bar{\gamma}^{i+1}, \sigma^{i+1})] \cdot \frac{1}{2} 2^{-\ell_1} \cdot \prod_{j=i+1}^{n} \frac{1}{2} \cdot 2^{-(\ell_1 + \gamma_{\sigma^i(1)})}$$

$$= \sum_{\ell_1=0}^{\infty} \Pr[A(\bar{\gamma}^{i+1}, \sigma^{i+1})] \cdot 2^{-(\ell_1 + 1 + (n-i)(\ell_1 + \gamma_{\sigma^i(1)} + 1))}$$

$$= 2^{-1+(n-i)(\gamma_{\sigma^i(1)}+1)} \cdot \Pr[A(\bar{\gamma}^{i+1}, \sigma^{i+1})] \sum_{\ell_1=0}^{\infty} (2^{-(n-i+1)})^{\ell_1}$$

$$= \frac{2^{-1+(n-i)(\gamma_{\sigma^i(1)}+1)}}{1 - 2^{-(n-i+1)}} \cdot \Pr[A(\bar{\gamma}^{i+1}, \sigma^{i+1})].$$

Moreover, it is clear that $\Pr[A(\bar{\gamma}^n, \sigma^n)] = 1$. Then noting that $\sigma^i(1) = \sigma(i)$, the solution is trivial:

$$\Pr[A(\bar{\gamma}^1, \sigma^1)] = \prod_{i=1}^{n-1} \frac{2^{-(n+1-i)-(n-i)\gamma_{\sigma(i)}}}{1 - 2^{-(n+1-i)}} = \frac{2^{-(\binom{n+1}{2}-1)}}{\prod_{i=1}^{n-1}(1 - 2^{-(n+1-i)})} \cdot \prod_{i=1}^{n-1} 2^{-(n-i)\gamma_{\sigma(i)}}.$$

Finally, plugging these terms into the overall probability of disjointness yields the expression in the theorem. We will use this expression in the next section to calculate the probability of bug manifestation. □

6. JOINING THE MODELS

We have now described the two fundamental random processes of our work. Though the two are interesting in isolation, it is by combining them that we will achieve our overall goal: to characterize the probability of the canonical data race manifesting, under various memory models.

Our first observation is to note that Corollary 5.2 can be further simplified, provided the segment lengths are drawn from a distribution with a very weak condition.

THEOREM 6.1. *Let $\bar{\Gamma} = \Gamma_1, \ldots, \Gamma_n$ be a distribution over segment lengths, drawn from $\mathbb{N}^n$. Assume that the marginal distribution of each segment length is identical (i.e., $\Gamma_i \sim \Gamma_j \ \forall\ i \neq j$); they needn't be independent. Then all permutations of segment shifts are equivalent, and*

$$\Pr[A(\bar{\Gamma})] = c(n) \cdot 2^{-\binom{n+1}{2}} \cdot n! \cdot \mathbb{E}_{\bar{\Gamma}}[\prod_{i=1}^{n-1} 2^{-i\Gamma_i}].$$

The proof is given in the full paper. Because the identicality condition holds for the critical window size, the theorem gives an indication of how it is that we can analyze the overall bug manifestation concretely. Recall that the process of Section 4 generates a uniformly random program of STs and LDs, then randomly "settles" each instruction in turn, according to the rules of the memory model. The process of Section 5 applies a random "shift" to a series of line segments, the key event for which is the mutual disjointness of all the segments. We now combine these two processes by letting the line segment lengths of the shift process be distributed as the critical window size of the settling process. An important subtlety is that *we generate a single initial random program, then independently reorder n copies of this program.* Though this makes the analysis more complex, it adds a degree of realism: with n identical threads, it is more natural that the same data race would be present in the same position of every pair of threads. The following two theorems summarize our key results.

THEOREM 6.2. *For $n = 2$ threads, the probability that the canonical data race does not manifest is the following, in each of the three main models.*

Sequential Consistency:	$\Pr[A] \approx 0.1666$
Total Store Order:[4]	$0.1369 > \Pr[A] > 0.1315$
Weak Ordering:	$\Pr[A] \approx 0.1296$

THEOREM 6.3. *As n grows, the probability of successful execution is identical in* all *models, up to lower order terms in the exponent. In particular,* $\Pr[A] = e^{-n^2(1+o(1))}$.

The first tightly bounds the probability of successful execution for the case of $n = 2$ threads; the second gives an asymptotic bound on this probability for large n. We leave the proofs of these theorems to the full paper. Both proofs are rather technical and build upon the theorems of the previous two sections. The only surprising observation necessary is that, when lower bounding a certain expectation over the critical window for n threads, it suffices to use only a single term of this expectation. Doing so achieves the asymptotic behavior we seek.

Key Observations: Interpreting Theorems 6.2 and 6.3 yields remarkable insights. Though the case of $n = 2$ substantively distinguishes the memory models, we find that as n grows, the probability in all memory models approaches the same value, up to lower order terms in the exponent. This dichotomy is a fundamental take-away for informing computer architecture decisions. Though the use of weaker memory models does increase the risk of program error, as the number of threads grows this risk grows negligibly compared to growth of risk of error in even sequential consistency. This is of particular importance given the trends towards ever larger multicores that enable more and more concurrent threads.

7. DISCUSSION

Limitations and possible extensions: Our analysis assumes that the program consists solely of loads and stores, when real programs include synchronization, arithmetic, etc. These instructions can affect the timing of the program, introduce data dependencies that limit reordering, or disallow certain types of reorderings. An important item for future work is to include acquire/release fences, which are necessary to simulate memory models such as Release Consistency [11]. These fences act as one-way barriers, allowing instructions to reorder into, but not out of, a critical section. This behavior can be easily modeled using settling (§3.1.2). Fences

[4] A very similar analysis achieves a similar result for Partial Store Order (PSO). We omit the result for brevity.

make concurrency bugs less likely to manifest, as programs with fences have fewer legal reorderings. However, we conjecture that adding fences will not significantly change the main conclusions derived in this paper.

Optimized implementations of SC: Our model of Sequential Consistency assumes a relatively simple implementation wherein each processor executes only one memory instruction at a time. Many SC implementations use aggressive optimizations such as speculative execution to compete with the performance of weaker memory models [10, 12, 7]. We do not consider this simplifying assumption to be a weakness of our model; rather, we believe our results about weak memory models can be extended to address optimizing implementations of strong memory models. In other words, concurrency bugs are more likely to manifest in an implementation of SC that uses aggressive reordering than in a simple (and slow) implementation.

Generality of Results: In this paper, we propose and study one specific probabilistic process to model program execution and thread interleaving. Clearly, there are other plausible models that can be studied. Our intuition is that the results in this paper have a certain robustness with regard to changes to the parameters in our models as well as to changes in the model. However, future work is required to formally validate this conjecture.

8. CONCLUSION

With the ubiquity of multicore systems and the trend towards integrating every more cores on a single chip, multiprocessor programmability has become one of the key challenges in computer science. Even with improvements in programmability, we are likely to see an increase in software defects, given the inherent difficulty of concurrent programming. Memory consistency models are at the center of the programmability discussion, since they determine the memory access semantics of parallel programs. The debate over memory models has historically revolved around the tradeoffs between programmability, performance and complexity. In this paper we bring a new axis to this discussion: *software reliability*. We study an analytical model and show that concurrency bugs are indeed more likely to manifest themselves in relaxed memory models, but surprisingly, that as the number of parallel threads increases, the difference between harsh and weak memory models diminishes. The latter observation can have important consequences on system designers when developing new memory models.

9. REFERENCES

[1] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, December 1996.

[2] S. V. Adve and M. D. Hill. Weak ordering—a new definition. In *Proc. of the 17th International Symposium on Computer Architecture (ISCA)*, 1990.

[3] AMD Corp. *AMD64 Architecture Programmer's Manual - Volume 2: System Programming*, July 2007.

[4] C. Artho, K. Havelund, and A. Biere. High-level data races. *Journal on Software Testing, Verification & Reliability*, 13(4):220–227, 2003.

[5] Arvind and J.-W. Maessen. Memory model = instruction reordering + store atomicity. In *Proc. of the 33th International Symposium on Computer Architecture (ISCA)*, 2006.

[6] H.-J. Boehm and S. V. Adve. Foundations of the C++ concurrency memory model. In *Proc. of the 29th Conference on Programming Language Design and Implementation (PLDI)*, 2008.

[7] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. BulkSC: Bulk enforcement of sequential consistency. In *Proc. of the 34th International Symposium on Computer Architecture (ISCA)*, 2007.

[8] M. Dubois, C. Scheurich, and F. A. Briggs. Memory access buffering in multiprocessors. In *Proc. of the 13th International Symposium on Computer Architecture (ISCA)*, 1986.

[9] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Proc. of the 24th Conference on Programming Language Design and Implementation (PLDI)*, 2003.

[10] K. Gharachorloo, A. Gupta, and J. Hennessy. Two techniques to enhance the performance of memory consistency models. In *International Conference on Parallel Processing*, 1991.

[11] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proc. of the 17th International Symposium on Computer Architecture (ISCA)*, 1990.

[12] C. Gniady, B. Falsafi, and T. N. Vijaykumar. Is SC + ILP = RC? In *Proc. of the 26th International Symposium on Computer Architecture (ISCA)*, 1999.

[13] J. R. Goodman. Cache consistency and sequential consistency. Technical Report 1006, University of Wisconsin-Madison, 1989.

[14] Intel Corp. *Intel 64 and IA-32 Architectures Software Developer's Manual—Volume 3A: System Programming Guide, Part 1*, December 2009.

[15] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, September 1979.

[16] J. Lee and D. Padua. Hiding relaxed memory consistency with compilers. In *Proc. of the 9th Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2000.

[17] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes—a comprehensive study on real world concurrency bug characteristics. In *Proc. of the 13th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[18] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *Proc. of the 32th Symposium on Principles of Programming Languages (POPL)*, 2005.

[19] SPARC International, Inc. *The SPARC Architecture Manual—Version 8*, 1992.